CS4605 Lab 5
Winter 2004

George W. Dinolt

March 4, 2004

1 Introduction

This lab is a demonstration of the application of the *current access set*. The idea is that the `transform` function only adds to the current access set. We also illustrate how to manage the sequence functions that appear in the Goguen/Meseguer paper. We are not (repeat not) illustrating any *non-interference* models in this lab.

We are also providing another illustration of the structuring and mapping capabilities of PVS using the `IMPORTING` command. In this case, we have the following structure. Here, the `system` specification describes the details of the implementation. We haven't specified any of the details in our case, but we could have. The `AccessState` specification describes how we view *State* in our model. Note that the system doesn't have an "explicit" notion of state. That is something that we impose. The state specification defines what we mean be state changes (transform) and what we mean by secure

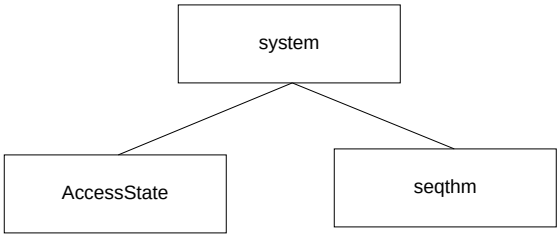

Figure 1: Structure of the specification

state. I have included the specifications at the end of this file for you to look at.

The `seqthm` specification shows how sequences of state changes are described. In the previous work, we defined a single sequence and assumed that it was constructed correctly. In this work we follow the model described in the non-interference approach. That is we start with a single initial state, and handle any sequence of inputs. We show that for any sequence of inputs, if the transform function has the "correct" properties, then every output sequence of states will be secure.[1]

2 What you need to do

You can find copies of the three specification here, AccessState.pvs,

system.pvs and seqthm.pvs. You can also find the specifications on `proof` at `/disk1/cisr/pvs-examples/lab5`.

You should download all three specifications into an empty directory and proceed to parse, generate tccs and prove any tccs, lemmas and theorems (but not `ASSUMPTIONS`) in the two specs. The `seqthm` specification has one longish theorem to prove. With enough expansions, etc. you should be able to manage it.

Once these two are handled, you should parse and generate the tccs for the `system` specification. Note that there are no theorems, but that if you prove all the tccs you get the theorem from `seqthm` for free.

You should pay attention as to how the mappings are accomplished. You should note that the `State` is not explicitly defined anywhere in the text of the `system` specification. It is imported from `AccessState`. The structure of `State` is imposed on the `system` specification. This is also true for other parts of the specification.

You should prove all the tccs. It turns out that there really is only one that requires any work. It looks very ugly. All the extra notation is to help you see which parts of the imported specifications the various types come from. You can ignore the extra detail. It does prove with the simple commands that you have at your disposal. You cannot (repeat **cannot**) use `grind` or its relatives. I want you to try and work out the details for yourself. This may help you see the relationships better.

[1] We might also want to show that such a sequence also satisfies a non-intereference claim, but that is too hard for this lab, i.e. I haven't been able to state and prove it yet.

You should provide me evidence that you were successful. That is, whatever you hand in should convince me that there were no holes left in the things that needed to be proved to accept and use the specification of `system`.

Good Luck.

3 The specifications

3.1 The "top level" system

```
system: THEORY
 BEGIN

  Subjects: TYPE+

  Objects: TYPE+

  Labels: TYPE+ FROM nat

  Modes: TYPE = {read, write}

  sl_sub(sb: Subjects): Labels

  sl_ob(ob: Objects): Labels

  IMPORTING AccessState[Subjects, Objects, Modes, read, write, Labels, sl_sub, sl_ob]

  IMPORTING seqthm[State, s0, Accesses, secure?, transform]

 END system
```

3.2 The State Definition in AccessState

```
AccessState[Subjects: TYPE+, Objects: TYPE+, Modes: TYPE+, read: Modes, write: Mo
bels:
            TYPE FROM nat, sls: [Subjects → Labels], slo: [Objects → La-
bels]]: THEORY
 BEGIN

  Accesses: TYPE+ = [# sb: Subjects, ob: Objects, m: Modes #]

  State: TYPE+ = setof[Accesses]

  s0: State = emptyset

  transform(a: Accesses, st: State): State =
      IF (m(a) = read ∧ sls(sb(a)) ≥ slo(ob(a))) ∨ (m(a) = write ∧ sls(sb(a)) = slo
        THEN (st ∪ {a})
      ELSE st
      ENDIF

  secure?(st: State): bool =
      ∀ (a: Accesses):
        st(a) ⇒
          (m(a) = read ∧ sls(sb(a)) ≥ slo(ob(a))) ∨
            (m(a) = write ∧ sls(sb(a)) = slo(ob(a)))

END AccessState
```

3.3 The Sequencing Structures in seqthm

```
seqthm[State: TYPE, s0: State, Inputs: TYPE+, st?: [State → bool], transform:
        [Inputs, State → State]]: THEORY
 BEGIN

  ASSUMING
   s0_secure: ASSUMPTION st?(s0)

   transition_state_secure: ASSUMPTION
     ∀ (st: State): st?(st) ⇒ (∀ (x: Inputs): st?(transform(x, st)))
  ENDASSUMING

  InSeqs: TYPE+ = sequence[Inputs]

  do(inseq: InSeqs, n: nat): RECURSIVE State =
    IF n = 0 THEN s0 ELSE transform(inseq(n − 1), do(inseq, n − 1)) ENDIF
     MEASURE n

  dobar(inseq: InSeqs): sequence[State] = (λ (n: nat): do(inseq, n))

  seq_is_secure: THEOREM ∀ (inseq: InSeqs): every(st?, dobar(inseq))

 END seqthm
```